Introducere in Design Patterns

Ce reprezinta design patterns? Ele reprezinta niste solutii generale, care pot fi reutilizate, ale unor probleme des intalnite in cadrul software design. In principiu, un design pattern este de fapt o schema a unei solutii pentru o problema de design (nu reprezinta un algoritm sau o bucata de cod ce poate fi aplicata direct intr-un cod-sursa), care poate usura munca unui programator si care poate duce la simplificarea si eficientizarea arhitecturii unei aplicatii.

Exista 3 tipuri de design patterns, in functie de aspectul si de functionalitatea obiectelor:

1) Behavioral Patterns: se trateaza modul de interactionare intre obiecte (Observer, Strategy, Visitor, Command)

2) Creational Patterns: se trateaza modul de creare a obiectelor (Factory, Singleton)

3) Structural Patterns: se trateaza modul cum este reprezentat un obiect, cum sunt relatiile intre entitati (Decorator, Composite, Proxy, Facade, Adapter)

Creational Patterns

Singleton Pattern

Descriere

Uneori ne dorim sa avem un obiect care sa apara doar o singura data intr-o aplicatie, de exemplu conducatorul unei tari (deoarece nu are sens sa existe mai multi in acelasi context). De aceea folosim Singleton, un mod prin care restrictionam numarul de instantieri a unei clase, ea avand o singura referinta ce va fi folosita in intreg proiectul. Astfel, pentru a asigura restrictionarea, clasa de tip Singleton va avea un constructor privat (il facem privat prin a bloca instantierea multipla a clasei), care va fi apelat in interiorul clasei. Pentru a putea crea o instanta unica a clasei (prin care se asigura o economisire a memoriei), vom avea nevoie de un membru static si privat, care va avea acelasi tip cu clasa pe care vrem o sa instantiem, si o metoda statica si publica prin care este returnata instanta clasei (se va crea una noua in caz ca nu exista deja una - lazy instantiation - una dintre cele 2 variante de instantiere Singleton, cealalta fiind eager instantiation).

```
public class SingletonClass {
    /*
    la inceput, inainte de prima si singura instantiere a clasei SingletonClass
    va avea valoarea null
    */
    private static SingletonClass obj = null;
    public int value = 10;
    // lasam constructorul clasei privat pentru a nu fi accesat din exterior
    private SingletonClass() {
        // do stuff
        System.out.println("Instantiam!");
    }
    // metoda prin care se creaza unica instanta a clasei
    // lazy instantiation
    public static SingletonClass getInstance() {
        // daca clasa nu a fost instantiata inainte, o facem acum
        if (obj == null)
            obj = new SingletonClass();
        return obj;
    }
    public void show() {
        System.out.println("Singleton is magic");
    }
}
```

Un avantaj il reprezinta faptul ca putem accesa metodele clasei de tip Singleton mai usor, neavand nevoie de instantiere sau sa dam un obiect care reprezinta instanta clasei ca parametru la o metoda (instanta clasei fiind un obiect vizibil la nivel global in cadrul unui proiect).

```
        public void modifyValue (int x) {
                SingletonClass.getInstance().value = x;
                // se modifica valoarea lui value din clasa
        }

        SingletonClass.getInstance().show();
```

Folosirea Singleton este dezavantajoasa in testare deoarece leaga dependente intre clase, ingreunand testarea acestora. Un alt dezavantaj il reprezinta folosirea acestuia in threaduri, nu este recomandat sa fie folosita varianta cu lazy instantiation in multithreading deoarece nu e thread-safe, in acest caz fiind recomandata folosirea variantei cu eager instantiation (mai multe despre multithreading veti afla la Algoritmi Paraleli si Distribuiti, in anul 3).

```
        public class SingletonClass
        {
            private static SingletonClass obj = new SingletonClass();
            private Singleton() {}
            // eager instantiation - merge la threaduri
            public static SingletonClass getInstance()
            {
                return obj;
            }
        }
```

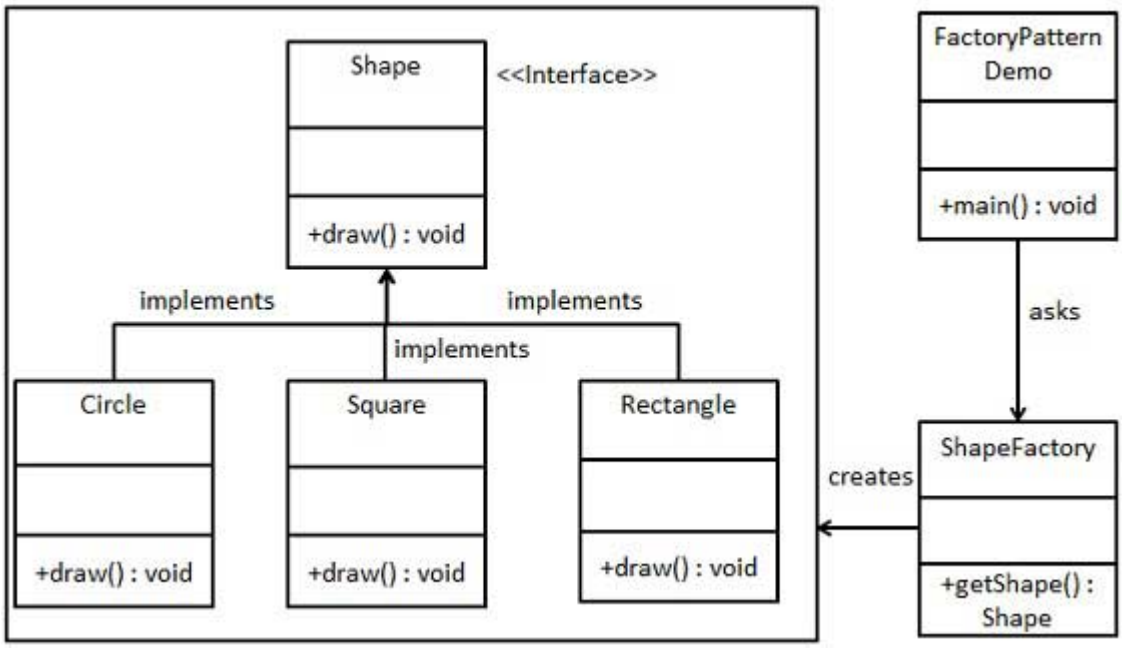
Utilizari

Utilizarile acestui pattern sunt: - inlocuirea de variabile globale (un obiect al instantei clasei de tip Singleton este global) - obiecte ce reprezinta surse partajate sau care sunt de tip logger, obiectele fiind accesate din mai multe locuri ale aplicatiei - implementarea de Factory (o sa vorbim mai jos despre acest subiect)

Factory

Descriere

Uneori suntem nevoiti sa cream obiecte in functiile de preferinta unui utilizator in cadrul unei aplicatii sau dupa alta necesitati. De aceea, folosim pattern-ul Factory, prin care alcatuim o familie de clase care sunt inrudite intre ele, prin faptul ca ele mostenesc aceeasi clasa abstracta sau ca implementeaza aceeasi interfata.



Astfel, prin exemplul de cod ilustrat mai jos, daca utilizatorul doreste sa afle informatii despre un tip de pizza, el va scrie la tastatura numele acelui tip de pizza si, daca tipul respectiv exista, el va primi informatii despre acel tip de pizza.

```
interface IPizza {
    void showStuff();
}
/*
 nu este neaparat sa avem o clasa abstracta ce implementeaza o interfata
 putem avea pur si simplu o clasa abstracta (fara sa implementeze o interfata)
 care e extinsa de clasele normale sau o interfata ce e implementata direct de
 clasele normale din Factory
*/
abstract class Pizza implements IPizza {
    public abstract void showStuff();
}
class PizzaMargherita extends Pizza {
    public void showStuff() {
        System.out.println("Sos tomat si branza Mozzarella.");
    }
}
class PizzaQuattroStagioni extends Pizza {
    public void showStuff() {
        System.out.println("Sos tomat, branza Mozzarella, sunca, pepperoni,
           ciuperci, ardei. ");
    }
}
class PizzaPepperoni extends Pizza {
    public void showStuff() {
        System.out.println("Sos tomat, branza Mozzarella, dublu pepperoni.");
    }
}
class PizzaHawaii extends Pizza {
    public void showStuff() {
        System.out.println("Sos tomat, branza Mozzarella, sunca, dublu ananas.");
    }
}
class PizzaFactory {
    public static Pizza factory (String pizzaName) {
        if (pizzaName.equals("Margherita"))
            return new PizzaMargherita();
        if (pizzaName.equals("Hawaii"))
            return new PizzaHawaii();
        if (pizzaName.equals("Quattro Stagioni"))
            return new PizzaQuattroStagioni();
        if (pizzaName.equals("Pepperoni"))
            return new PizzaPepperoni();
        return null;
```

```
        }
    }
```

Singleton Factory

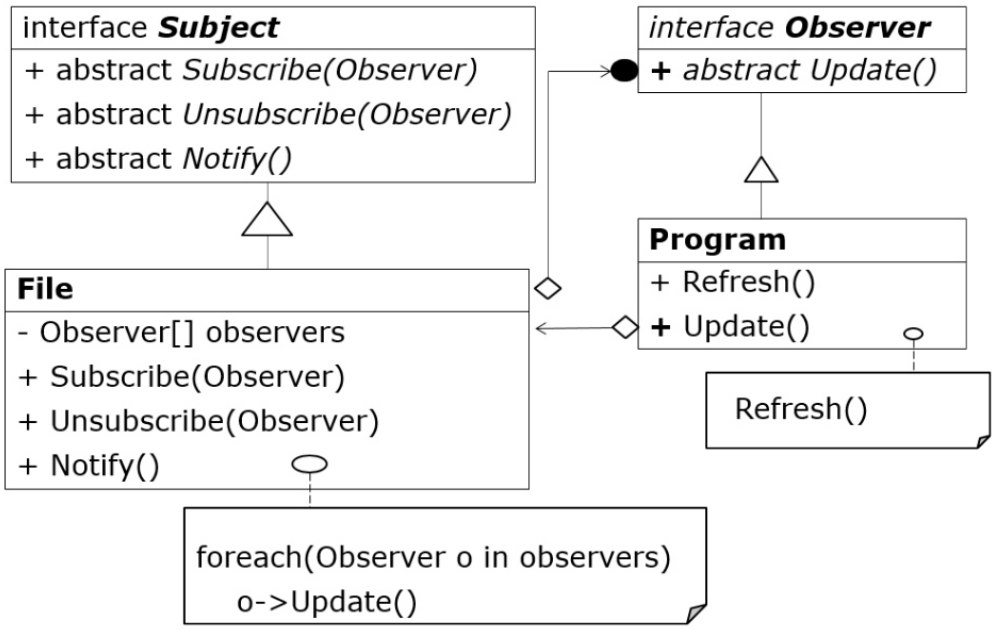
O clasa de tip Factory poate fi utilizata in mai multe locuri in cadrul unui proiect si pentru a economisi resurse putem folosi pattern-ul Singleton, existand o singura instanta a clasei Factory.

Behavioral Patterns

Observer Pattern

Descriere

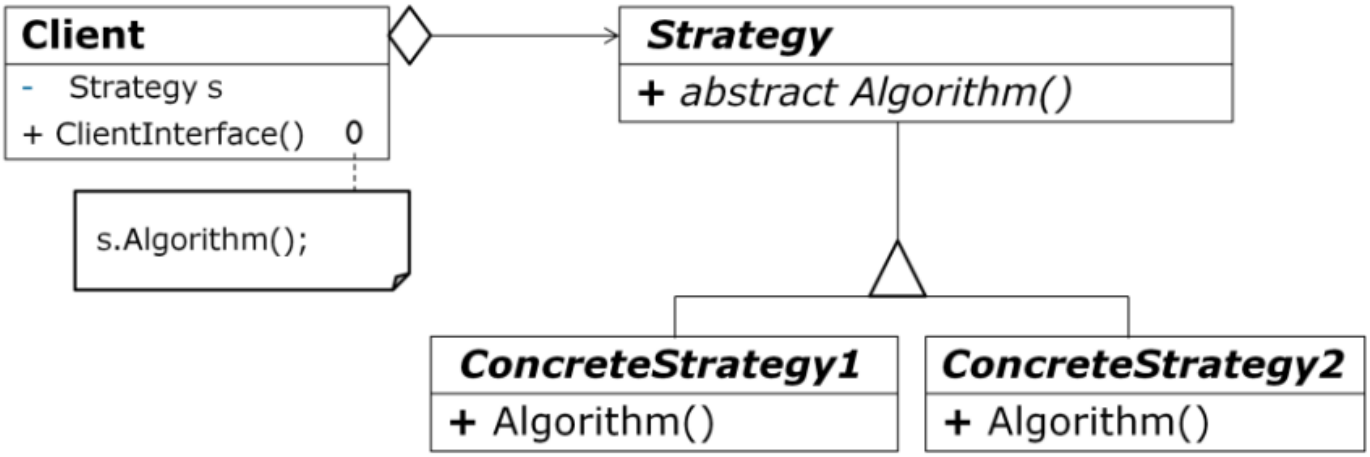
Acest design pattern stabileste o relatie one to many intre obiecte. Mai exact, avem un obiect pe care il numim subiect, caruia ii este asociat o colectie, o lista de observatori, care sunt obiecte dependente de subiect, notificate in mod automat de catre subiect, cand in cadrul subiectului are loc o actiune sau o modificare a starii subiectului.



Strategy Pattern

Descriere

Strategy reprezinta un design pattern behavioural ce ofera o familie de algoritmi, adica de strategii, incapsulate in clase ce ofera o interfata de folosire (in cazul exemplului de mai jos - intefata Strategy), din care clientii / utilizatorii pot sa aleaga. Acest design pattern este recomandat daca este nevoie de un tip de strategie / algoritm cu mai multe implementari posibile si se doreste sa se aleaga in mod dinamic un algoritm pentru a-l folosi. De exemplu, daca ne dorim sa cautam un element intr-o colectie putem sa vedem mai intai daca este sortata colectia sau nu, ca sa vedem ce algoritm de cautare folosim. Daca este colectia sortata, putem aplica algoritmul de cautare binara in colectie, altfel putem sa facem o simpla iterare prin colectie ca sa vedem cautam elementul dorit.

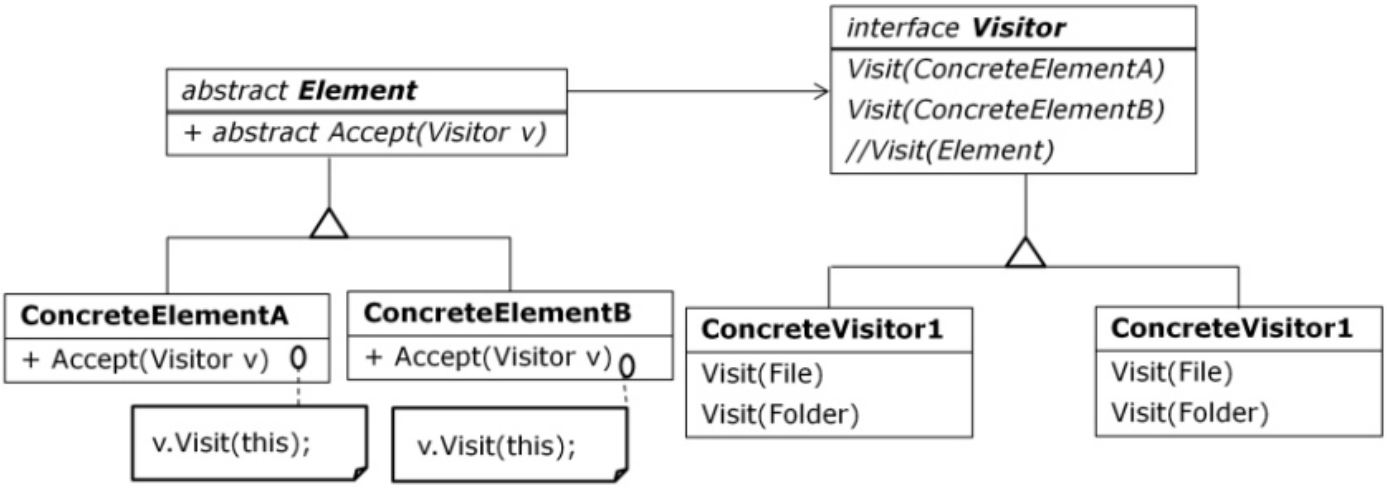


Visitor Pattern

Descriere

Acest design pattern ofera posibilitatea de a separa un algoritm de structura de date pe care acesta opereaza, astfel noua ne este usor sa adaugam functii noi care opereaza peste o familie de clase, fara sa modificam structura acestora. Pe scurt, folosim Visitor daca avem tipuri diferite si dorim sa adaugam sau sa schimbam operatii fara sa modificam clasele.

In cadrul acestui design pattern, avem o interfata Visitor, care reprezinta operatia aplicata, si o interfata / clasa abstracta Visitable (numita in imagine Element), care reprezinta obiectele pe care se aplicatii operatiile din clasele de tip Visitor. In clasele Visitor, vom avea o metoda visit, prin care Visitorul "viziteaza" un obiect tinta, care este reprezentat ca fiind de tip Visitable, o clasa de tip Visitable continand o metoda accept, care accepta "vizita" Visitor-ului asupra sa.



Un exemplu de Visitor este implementarea de operatii pentru fisiere precum cat si ls. In cadrul acestui exemplu, vom avea ca Visitori clasele Ls si Cat, iar ca Visitable vom avea clasele Fisier si Folder, care mostenesc o clasa abstracta numita Repository. Vom exemplifica mai jos, pentru o mai buna intelegere:

```
        interface Visitor {
            void visit (Director f);
            void visit (Fisier f);
        }
        class Ls implements Visitor {
            public void visit (Director f) {
                System.out.println(f.getName());
                for (Repository repo: f.getChildren()) {
```

```
                    System.out.println("\t" + repo.getName());
                    // afisam numele unui repo (fisier / folder)
                }
            }
            public void visit (Fisier f) {
                System.out.println("Not a folder");
                /* comanda Ls (in acest exemplu) este specifica doar folderelor,
                in acest caz este evidentiat un dezavantaj al  Visitor-ului,
                faptul ca noi trebuie sa implementam metode de care nu avem nevoie
                in acest caz - se incalca Interface Segregation Principle  */
            }
        }
        class Cat implements Visitor {
            public void visit (Director f) {
                // avertisment ca avem folder, nu fisier
            }
            public void visit (Fisier f) {
                // citire fisier, folosind numele fisierului
            }
        }
        abstract class Repository {
            private String name;
             // numele unui fisier sau folder (de fapt, calea acestuia)
            public String getName() {
                return name;
            }
            public abstract void accept (Visitor f);
        }
        class Fisier extends Repository {
            public void accept (Visitor f) {
                f.visit(this);
                // Visitor-ul "viziteaza" fisierul, adica acesta
                //efectueaza o operatie asupra fisierului
            }
        }
        class Director extends Repository {
            private List<Repository> children = new ArrayList<>();
            public List<Repository> getChildren() {
                return children;
            }
            public void accept (Visitor f) {
                f.visit(this);
            }
        }
```